Performance Evaluation of High-Level Language Systems

Bruce Leverett

DEPARTMENT of COMPUTER SCIENCE

Carnegie-Mellon University

Approved for public release;
distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 AFOSR/TR-78-1088 — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 PERFORMANCE EVALUATION OF HIGH-LEVEL LANGUAGE SYSTEMS.	5. TYPE OF REPORT & PERIOD COVERED 9 Interim rept. 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Bruce/Leverett	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213 17 07	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101E 16 2466 A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209	12. REPORT DATE 11 Nov[redacted]977 13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report) UNCLASSIFIED 12 42p. 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release' distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

One of the concerns of the compiler writer is the quality of object programs produced by the compiler, and in particular theri performance at execution time. A survey of methods for measuring this performance, and experiments with the use of those methods, is presented. We examine two general categories of evaluation: comparative evaluation, in which benchmark programs are run on groups of language systems; and analytic evaluation, in which a singel system is measured in terms determined by its own structure. Besides surveying the results of various evaluation experiments, we present in detail the results of a series of experiments on a particular language system (PDP11 ALGOL 68-S).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

2

Performance Evaluation of High-Level Language Systems

Bruce Leverett

November 1977

Abstract

One of the concerns of the compiler writer is the quality of object programs produced by the compiler, and in particular their performance at execution time. A survey of methods for measuring this performance, and experiments with the use of those methods, is presented. We examine two general categories of evaluation: comparative evaluation, in which benchmark programs are run on groups of language systems; and analytic evaluation, in which a single system is measured in terms determined by its own structure. Besides surveying the results of various evaluation experiments, we present in detail the results of a series of experiments on a particular language system (PDP11 ALGOL 68-S).

This work was supported by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research.

Introduction

The overall performance of a computer system depends not only on the design and configuration of the system itself, but also on the nature of the programs which run on it. If, as with many systems, almost all user programs are processed by one of two or three compilers before being run, changes in the quality of object programs produced by those compilers can have noticeable effects on system performance. Any means of assessing or analyzing the performance of object programs, therefore, can be extremely useful to the compiler writer. This paper surveys the efforts which have been made at doing such analysis.

Most language processing systems consist of two phases: a "translator" and an "interpreter". The interpreter may be fairly close to the instruction interpreter of the computer on which it runs, in which case we call the translator a compiler. In any case the performance of the system as a whole depends on three factors: the performance of the translator itself; the performance of the interpreter itself; and the quality of the translation, i.e. the degree to which the program output by the translator makes the best possible use of the interpreter. For most systems, either the first of these factors is very important, or the last two are, but not all three. We will concern ourselves with the last two factors, i.e. with systems in which most of the processor time spent on a program is in executing the translated program.

Except for assembler systems, it is almost never true that the interpreter is exactly the instruction interpreter of the computer. The difference between the "virtual computer" for which the translator generates code, and the "real computer" which it may resemble, is defined partly by the set of utility subroutines used in the interpreter (e.g. subroutines to do input/output, or to do dynamic storage allocation), and partly by a set of conventions which are enforced and obeyed by the code output by the translator (e.g. subroutine linkage conventions, or register and memory allocation conventions). We will refer to this set of subroutines and conventions as the "run-time system" of the language system. One of the compiler writer's concerns is with the efficiency of the virtual computer so defined, relative to that of the real computer on which it runs.

As with other (hardware and software) systems, we can study the behavior of language support systems in two ways. If we directly compare the performance of one system with that of another with similar input, we are doing *comparative* evaluation of the systems; if we measure the performance of a system "on its own terms", without reference to other systems, we are doing *analytic* evaluation (see [1]). We will use these two categories to classify the performance measurement studies that we will discuss, because the methods and goals of the two types of evaluation are fundamentally different. We will see that many techniques have been used to try to get a variety of kinds of information about the behavior of language run-time systems; and we will try to impose order and direction on the resulting chaos.

1. Comparative evaluation

Detailed comparison between language systems, in the measures of execution performance, requires that identical "benchmark" programs be run on them. This is a bare minimum requirement; it often happens that even this does not suffice to allow meaningful comparisons between program runs:

- If the language systems run in different environments (different computers, or even different operating systems on the same computer), it is difficult to separate the effect of the environment on performance from the effect of the language system software. We shall see that some attempts have been made to do this by purely statistical means, i.e. to assign to every environment a set of multiplicative factors that describe its effect on the performance of various types of programming constructs. These methods are of limited accuracy and usefulness, however.

- Even within the same environment, completely different organizations of run-time action may render nearly meaningless the comparison of execution times for certain types of programs. Consider, for instance, the allocation of space for variables. On some ALGOL 60 systems, all allocation of local static storage for a procedure is done at procedure entry; on others, allocation of storage local to each block is done at entry to that block. Clearly the comparison of times required for block entry and exit between systems of the two different types is not very useful. Even for the simplest of benchmark programs, it is sometimes not possible to avoid comparing apples with oranges.

At the same time, the "bare minimum" requirement given above is not quite a minimum. Valid comparisons can be made between language systems which implement different languages; in this case the benchmark programs used will be different from each other, hopefully in small ways. Such comparisons are fraught with traps for the unwary:

- Different languages are apt to be similar in many ways but to have completely different capabilities in other ways. We will see examples of this later; it means that the transcription of a benchmark program from one language to another may not be straightforward, and may be a knotty problem indeed.

- Patterns of use of language constructs and features are dependent on the languages themselves. A program whose usefulness as a benchmark arises from its resemblance to a "typical" user program may lose its typicalness when transcribed to another language.

Finally we should note that the whole business of timing the execution of a program is not trivial and that a number of factors may render invalid times which are recorded in an incautious manner:

- Though most general-purpose computer systems include a clock with which to time program execution, and instructions or operating system calls to read or reset the clock, these clocks are of highly varying grain size. Some are (claimed to be) accurate to within a few microseconds; others cannot be relied on to within less than a second. If we are interested in the execution times of small sections of code, such as individual subroutines or even individual statements, we must often either arrange for the code to be executed thousands or millions of times in a loop, or rely on the computer system's published instruction timings and our knowledge of the machine code that is executed. We shall see that both these methods have their own disadvantages.

- The time taken for identical program runs on the same system may vary, especially on multiprogramming systems, because of phenomena such as cycle stealing for I/O transfers going on concurrently with the program, or interrupts, in either case due to activity not under control of the program being timed. Many if not most of the clocks available in various systems do not discount such lost time from the time recorded as taken by the program in question. One method of getting around this problem is simply to run the program several times, and use the smallest of the recorded execution times as the "official" one. But this is not an entirely satisfactory solution.

In spite of the formidable problems outlined above, several interesting comparative studies have been done, some involving several dozen language systems. We will examine the methodologies used in these studies, and in particular the design of the benchmark programs and the purposes to which they were directed.

1.1. Basic statements

Wichmann ([2],[3],[4]) uses a benchmark program which attempts to study at once a wide range of characteristics of Algol 60 systems. This is done by timing a set of "basic statements". A complete description of this method is in order since it is based on a view of programming language systems that is popular and has been widely used in other studies ([12],[15],[25]), in spite of its rather shaky validity.

Wichmann has designated a group of simple Algol 60 statements to be "basic". The complete list of these is found in figure 1 in Appendix A. An attempt has been made to make this set of statements complete, in this sense: that if one could get average times for each statement in the set for one particular system, one would completely understand the timing behavior of the system, i.e. one could make good estimates of the time taken for whole programs running on that system.

There are some obvious and probably deliberate omissions of Algol 60 features from the list, e.g. call-by-name parameters, own variables and own arrays. There were also some individual statements that were chosen poorly, as Wichmann points out; for instance, for the statement

```
e2 [1, 1] := 1
```

the address calculation necessary to do the array access was done at compile time in systems and at run time in others, causing an "apples vs. oranges" comparison between systems that could easily have been avoided by the use of non-constant subscripts. (Because of the large number of systems on which this experiment was done, it was not practical, for reasons of compatibility, to simply change the list of basic statements to correct such minor deficiencies).

There is a more fundamental problem, however, with the underlying assumption: that the action of the system during execution of a program can be divided neatly and charged to the actions of the separate statements. This is probably the case for many Fortran systems; but for systems implementing more sophisticated languages, including Algol 60, it frequently fails in significant ways to describe reality, and performance models based on this assumption are bound to be misleading.

The problem is not confined to the so-called "optimizing compiler" systems, in which statements are combined with one another or moved out of loops, and expressions may be discovered to be redundant and not calculated. (In fact Wichmann acknowledges the problems inherent in running the basic statement benchmark program on these systems, and has designed a different benchmark program in response to this issue; this is discussed in a later section). Even systems in which no optimization is done frequently carry on activities whose costs cannot be fairly assigned to particular statements or even groups of statements. For instance:

- In many systems, all the code for the declarations within a procedure is at the beginning of the procedure, even though declarations are allowed elsewhere. Also, some systems process groups of declarations at once, even rearranging the order of consecutive declarations, e.g. processing a group of declarations of integers, followed by a group of declarations of reals, etc. although the integer and real declarations were interspersed in the source program. Clearly it is not very useful to try to isolate the effect of a single declaration in such a system.

- In a system which includes a dynamic storage allocator at run time, the behavior of the allocator is seldom correlated strongly with statement boundaries and characteristics. For instance, coalescence of available free storage, or even compaction (by rearrangement in core) of storage blocks which are in use, are likely to be all-at-once, time-consuming operations which are performed at seemingly random, unpredictable points during program execution.

- Some systems attempt to avoid copying of large blocks of storage, such as arrays, back and forth by keeping track at execution time of how they are used. For instance, if an array is to be returned as the result of a procedure call, it may be possible to leave it in place on the control stack as the stack is popped, not copying it downward unless it proves necessary to do so ([5]). In this situation, the construct which causes the array to be copied (such as a subsequent procedure call) is not the construct to which the copying should be charged (i.e. the original procedure return), rendering more difficult the neat separation of the costs of constructs.

Nevertheless the "basic statements" benchmark program has become extremely popular and has been run on about 80 different Algol 60 systems (including a few Algol 68 and Pascal systems). The program times each separate basic statement by executing it in a loop; the number of iterations of the loop and the time-clock reading statements before and after it are supplied by the person who runs the program on any particular system.

The loop is ordinarily a for loop whose body is a single instance of the statement in question. This pattern may be modified for either of two reasons:

- The execution time of the statement may be comparable to, or even considerably less than, the execution overhead of the loop statement itself. In this case no number of iterations is large enough to give a reliable timing; to remedy this the loop body is changed to consist of several repetitions of the statement.

- "Optimizing compiler" systems may move the code generated for the basic statement out of the loop, or (in the case described above) recognize that all but the first of the repetitions of the statement are redundant, and not generate code for them. The system tester must find ad hoc ways of getting around these problems; for instance, most such systems allow optimization to be "turned off" over small sections of the program or even the entire program.

From the time taken for each loop, the time taken for the loop with a null body is subtracted, yielding an average execution time for the statement.

This raw data is interesting enough in itself, both because it can help to pinpoint weaknesses in the performance of a system, and because it can in a vague way give us an idea of the relative merits of different language system organizations and the principles (if any) on which they are based. Wichmann has done some further analysis of the data along statistical lines which, though admittedly somewhat unreliable, are interesting because of the large number of systems involved.

- Making the assumption that the time T_{ij} taken for statement i on system j is the product of two factors, one of which depends only on i and the other only on j, he computes the set of factors using a least-squares fit. Even more interesting than the factors themselves are the residuals R_{ij}, the ratios between the expected times based on the model and the actual times. Values of these which are greatly different from unity indicate, rather more clearly than simple examination of the raw data, which features of an implementation are unusually slow or fast relative to the implementation as a whole. Wichmann also investigates the pairwise correlation coefficients of the R_{ij}'s, arriving at graphs of correlations between statements and between systems, which are of value as a curiosity if they are not directly useful in aiding understanding of the systems involved.

- In an earlier study, Wichmann ([6]) gathered some statistics on the relative frequencies of execution of the various basic statements, enabling the direct comparison of the set of statement times from one system with those of another, by giving weights to the basic statements and forming a weighted average for each system. While it is not clear that this weighted average can be regarded as a valid measure of system performance, because of the objections raised earlier, or whether any meaning is left at all when all of the performance data about a system are squeezed into a single number, it is probable that this figure too can be used as an order-of-magnitude estimator of the "average" speed of programs running on a particular system.

Appendix A presents the results of running the basic statements benchmark on a particular system, CMU Algol 68, with some discussion of the relevance of the program to Algol 68, and the characteristics of the system which are brought to light by the data.

1.2. Procedure calling overhead

The very same Wichmann mentioned in the previous section reports ([7],[8]) on another very popular benchmark program, intended to provide insight into the overheads of procedure call and return in Algol-like languages. The program is a version of Ackermann's function which has a number of advantages as a benchmark: it is small; it uses almost no interesting features of the language other than the procedure calling mechanism; its behavior, in terms of the required stack size, the number of procedure calls that take place, and the relative frequencies of execution of each conditional branch of the code, has been exactly characterized ([9]) in terms of the parameters passed to it, making the study of its behavior much more straightforward.

Wichmann presents four figures for each program run: the time taken per call (in microseconds); the average number of instructions executed per call (this was deduced from object code listings); the number of words of stack growth per call; and the size in bytes of the object code for the whole procedure. In this case evidently the actual running of the program, to obtain the first figure, is of considerably less interest than the analysis of the object code to discover just why the four figures are what they are. This is not surprising in view of the wide variety of procedure calling mechanisms used; to extend the metaphor of "apples vs. oranges", the 60 systems on which this program was run represent, not 60 varieties of apples, but perhaps 60 different species of fruit. The calling conventions can be categorized along at least five dimensions, as described by Wichmann:

- Nature and extent of stack overflow checking. Systems may check this in software at procedure entry, or place the stack so that there will be a hardware trap when it exceeds its limit. Of the former systems, some allow for dynamic storage allocation by performing garbage collection if the stack overflows.

- Environment setup at procedure entry. Some languages allow addressing of only local variables and static (global) variables; some allow addressing of non-local non-static variables and thus require maintenance of a display. (Also, there are many possible implementations of a display).

- Dynamic stack storage. Languages which allow declaration of storage whose size is not known until execution (e.g. dynamic arrays) require two stack pointers to be maintained; more restricted languages require only one.

- Parameter passing conventions. In some systems, all parameters are passed by reference; in others, the parameters are passed by value (all of them can be so passed to this procedure). Some systems created thunks for the parameters anyway.

- Library subroutines. Various low-level support functions, up to and including the entire procedure call/return mechanism, are coded as calls to library subroutines by various different systems.

In view of the qualitative differences between even the best systems it is remarkable that comparisons of any interest can be made between them; but in fact the comparisons of various systems all running on the same machine are fascinating and instructive and -- who knows? -- perhaps provided useful guidance to the implementors of some of the systems tested.

1.3. Other studies

Boom and de Jong ([10]) used several different benchmark programs, to compare six systems involving four different languages (Pascal, Algol 68, Algol 60, and Fortran) on the same computer (the CDC Cyber 73). In addition to the two benchmark programs by Wichmann, they used two programs of their own devising, which we will comment on.

The first of these was a program to symbolically compute and print out the first hundred and fifty cyclotomic polynomials. It is difficult to deduce from the report just why this program was selected as a benchmark. The authors give one clue: that it is "... a real program, at least one version of which had been originally written for a purpose other than that of testing the compiler". There are other peculiarities of the program that make it suitable for the measurements which the authors had in mind. For each program run, they measured the CP (central processor) time required for the whole run, the CP time required for the calculation of coefficients of the polynomials, and the CP time required for formatting and printing of the polynomials. We suspect that one of the reasons the program was chosen is that it is organized so that these last two times are easily separable. (The first half of the program computes the coefficients, and the second half formats and prints them). (Other measurements were made, such as the CP time required to compile the program, but these were not measurements of the run-time system and are not of direct interest to us).

The program was run several times on each system, once for each possible combination of compiler options which could affect performance. The most common option available was for subscript checking of array accesses, but some of the systems had an additional option involving miscellaneous object code optimizations as well. It seems that array subscript checking is not done in the same way by all the different systems:

- Some of them check that every subscript in a multidimensional access is within its bounds;

- Some of them check only the offset calculated from all the subscripts and dimensions of the array; if adding this to the address of the base of the array produces an address that is within the array, the access is deemed legal;

- One of them does an optimization on subscript checking: if the subscript is the index of a do loop, the checking is done on the bounds of the do loop at loop initialization time rather than when the array is actually accessed.

In this case the principal barrier to comparability of the results was the differences between the languages. Four different versions of the program were used, of course. The authors' philosophy in writing versions for the different languages was that every attempt should be made to take advantage of the characteristics of each language. (The opposing philosophy is that the versions should look as much like each other as possible). This is a user's idea of comparability rather than an implementor's; it compares the costs, on the different systems, of doing some particular task, rather than of using some particular common language feature. However, in certain cases the authors' attempt to take advantage of language features has had the opposite of the intended effect, i.e. they have taken advantage of a feature the use of which increases program clarity or naturalness but degrades performance:

- The Algol 68 version uses flexible arrays for the coefficients, where the other versions use arrays which are fixed in size at the maximum. This means that the Algol 68 version may save core storage (if the system can give back the unused storage to the operating system, a highly unlikely possibility), but it pays a penalty in execution time.

- The Algol 60 and Algol 68 versions use variables local to the blocks in which they are used; Pascal and Fortran do not allow this. This may be beneficial for the performance of the Algol systems, but it may be detrimental, depending on the implementation and on the relative frequencies of entering local blocks compared with entering procedures.

- The Fortran version uses formatted I/O. Due to the necessity of interpreting formats at execution time, Fortran formatted I/O is well known to be a source of performance problems. For some reason the authors chose not to use formatted I/O in the Algol 68 version.

- On the other hand, since every output statement in Fortran starts a new output record, if several numbers are to be output in one record, they must all be output in one statement. Thus the authors have a buffer in the Fortran version to hold a line's worth of coefficients, and the whole line is output at once; in the other systems each coefficient is output separately. Thus a deficiency in Fortran leads to a performance benefit for the Fortran version.

- In the Algol 68 version, the procedures which multiply and divide polynomials are represented as operators which take arrays as arguments and return arrays as values. The other systems do not allow procedures to return arrays as values. Unfortunately many systems do not make any effort to avoid copying arrays back and forth when they are passed as parameters or returned as procedure values, and so this practice may cause grave performance problems with such systems ([23]). The authors, apparently recognizing this problem, have coded the operators to pass around references to arrays rather than the arrays themselves, but the extra level of indirection at every array access somewhat degrades performance as well.

This illustrates the pitfalls involved in even the most reasonable and conscientious policy of transcription from one language to another. Nevertheless the performance figures arrived at are illuminating, even considering that the benchmark program may not in any way be representative of most user programs.

The authors use another benchmark task for more extensive testing of I/O performance; in this case the·program simply copies the first 200 lines of one file into another. There are several very different ways to do this on each system and each way was tested; for instance:

- There are three Algol 68 versions: one does character at a time I/O, another does line at a time I/O, and the third "pretends" to do character at a time I/O but internally keeps buffers and does string I/O, a technique which a casual user of the Cyber 73 Algol 68 system might use because of the high overhead associated with every call of the system I/O.

- There are two Fortran versions: one uses format 80A1 to read lines of eighty characters; the other uses format 8A10.

This is strictly a user's comparison; no attempt is made to further analyze the results, or to take any especial care in making the programs identical.

Curnow and Wichmann ([11]) have attempted to address some of the weaknesses of the basic statements benchmark by designing a "synthetic benchmark" program (see [12]) -- a program carefully designed so that its requirements for various system services matched those measured for the average workload of a system. In this case the "system services" were defined in terms of the types of interpreter instructions in the intermediate-level code generated by the Whetstone ([13]) system. The problems attacked by this program are:

- The structure of the basic statements program was such that an "optimizing compiler" could render its measurements useless. The synthetic benchmark is coded carefully to be almost immune to the classical optimizations of flow analysis; this is probably bending over too far backwards, however, because most user programs of moderate or greater complexity are affected in their performance when these optimizations are performed. More importantly, the basic statements benchmark could be rendered less useful by a compiler which took advantage of its simple structure to do most computations in fast registers; it is hoped that the more "natural" appearance of the synthetic benchmark will result in register allocation being more normal in the code compiled for it.

- Some of the basic statements, such as the array accesses using constant subscripts mentioned earlier, were unusually simple cases of the language features they were intended to represent. It was hoped that this would be corrected in the synthetic benchmark. Presumably, however, this would be no more beneficial than correcting them in the original basic statements benchmark.

- The authors say of the basic statements benchmark that "... it was not always easy to obtain in one program sufficiently accurate processor times [for the 42 basic statements] ..." It is not clear, however, that the synthetic benchmark corrects difficulties that may have arisen in this regard. A run of the program yields only one performance figure, a system speed in pseudo-instructions/second, rather than 42. The fact that all the inaccuracies of timing have been swept together does not mean that they have been reduced.

The synthetic benchmark is run on several systems on two computers, and is compared, as a measure of machine speed, with three other programs: a similar benchmark written in Fortran, the basic statements benchmark, and a Gibson mix ([14]) of instructions. The Fortran benchmark and the Gibson mix are shown to be closely correlated to each other, but not closely correlated with the other two, which are in turn closely correlated to each other.

1.4. Summary

We have seen that apples can, indeed, be compared with oranges. In fact, as the experience with the basic statements benchmark and the procedure calling benchmark has shown, implementors and maintainers of systems will go to infinite time and trouble to prepare their systems for comparison with other systems, no matter how little useful information they stand to gain from the comparison!

If we draw a distinction (also see [12]) between "user's" comparisons, which are intended to help potential users of systems choose between them or judge of their relative speeds, and "implementor's" comparisons, which assist the implementor of a system in pinpointing its strengths and weaknesses, we can better understand this phenomenon: the procedure call benchmark, which is poorly designed as an implementor's comparison, is ideal as a user's comparison because of its simplicity and narrowness of scope, and has therefore become phenomenally popular.

The future of implementor's comparisons may well be in doubt. We have seen that even in a group of fairly old Algol 60 implementations, the "basic statements" assumption was beginning to come under question because of optimizing compilers; more modern compilation techniques, including dynamic storage allocation and systems for avoiding the copying of large values, have made it even more difficult to find compartmentalizations of program execution costs that can be easily reflected as individual source language constructs or program fragments. Perhaps as the virtues of orthogonal language design become more widely recognized, and the use of these techniques in language systems becomes more routine, it will become much less useful to system implementors to try to compare the performance of such systems on a feature-by-feature basis.

2. Analytic evaluation

The criteria and methods for selecting programs to run in order to study a language system are different, when the system is being studied in its own terms, from what they are when it is being compared with other systems. We saw in the previous section that the central part of the design of a comparative study is the design of a "benchmark program", a single program which can be run identically or at least comparably on several systems. Non-comparative studies of language systems, on the other hand, have generally attempted to get data from runs of a wide variety of programs; these studies are really getting data on the workload faced by the system as well as on the system itself, and the more programs which can be run, the smaller the chance of getting a distorted picture of the workload due to the accidental peculiarities of a few programs. The criteria for choosing test programs are not always the same:

- Knuth ([15]) sought a group of programs which would be "typical" in some sense of the entire computing load of the Fortran system under study. The criteria by which typicalness was judged were

  - The average level of sophistication of the programs should not be too far from the universal average;

  - There should be programs written for many different applications in the sample. He did not attempt to control the proportions of each type of application included.

- Clark ([22]) sought a group of programs that would not be "average" in any sense, that is, programs distinguished for their largeness, complexity, and sophisticated use of list structure. The principal justification for this is that, if regular patterns of activity or accessing are found in the Lisp system's treatment of these programs, they will be applicable to smaller programs as well; whereas peculiarities or other patterns may be found in a set of smaller or "typical" programs, which disappear for the large and sophisticated applications. Note that Lisp is not the standard language for applications programming at many computer installations, and it is still more important to improve the performance of Lisp systems on large, sophisticated programs than on small ones; the Fortran system which Knuth studied, on the other hand, faced a daily workload in which large, sophisticated programs played a very small part.

Other authors were less explicit about the criteria by which test programs were selected; we suspect that the temptation to use whatever programs are handy is very strong. The nature of the set of sample programs depends to some degree on how they are collected:

- The most reliable method for getting a set of sample programs that is "typical" of system usage is to gather data on *every* program run on the system over some period of time. Unfortunately this is not usually possible. Many of the data gathering systems we will describe slow down the execution of programs by so much that it would not be acceptable to impose them on all users or even on a random selection of users of the system.

- Knuth describes a method of getting programs by "... rummaging around in wastebaskets and recycling bins". However, a disproportionate number of programs gotten this way are incorrect, indeed uncompilable. While this would not be a bad method of getting data on the distribution of types of errors, it is inappropriate for a study of programs which are actually executed.

- It is possible to solicit programs from their authors. This can be done as in [22] when there is a small group of potential authors of usable programs; or when there is a central facility, such as a card reader, which all programmers must use to submit programs to be run. This method has the advantage that incorrect programs can be weeded out, and if the programs are complex and have non-trivial requirements for input data, these can be described and documented by the programmers. Alexander ([17]) used a number of programs which were available because they had been submitted for a course in compiler writing.

  Note that with this method, and indeed with any method except the first one, we get only a group of test programs, with no information about how often they are run, either in absolute frequency or relative to each other. If we are interested in the average workload of a system, our idea of it is incomplete unless we have some sense of the relative proportions contributed by various types of programs.

- On computer systems in which programs can be stored in a file system for long periods of time, it is possible to rummage around in the file system to find suitable tests. This has the same disadvantages as rummaging around in wastebaskets, but in considerably less aggravated form, i.e. one is likely to find a larger proportion of programs which actually run.

- The programs most easily available are "system programs", e.g. compilers, or "classical benchmark" programs, e.g. programs from subroutine libraries, or programs used to test the correctness of the compiler or the run-time system. Of course, whatever their merits, these programs are unlikely to be typical in almost any respect of the workload presented to the system.

There is a spectrum of usage characteristics which affect system performance, from those which can be studied without the slightest reference to the organization of the language run-time system, to those which cannot even be expressed without drawing on the reader's familiarity, either with language run-time systems in general, or the particular system under scrutiny. We will examine a group of studies that span most of this spectrum, pointing out the data gathering techniques used in cases where they are novel, and describing the types of results which were obtained and how they could be put to use, although not in most cases the actual results themselves.

2.1. Knuth

Perhaps the best-known study of language usage patterns is by Knuth ([15]). Part of this is a study of static usage, i.e. of relative frequencies of features of source programs, rather than frequencies of events at execution time. There have been several more studies of this type (see [16]); as a rule they are of little interest to the subject of execution-time performance. However, many of the measurements reported by Knuth could have been extended to dynamic measurements; they were not, apparently only for lack of time. For instance:

- All Fortran statements can be classified by "statement type", determined by the keyword at the beginning of the statement (e.g. do, continue), except for assignment statements, which were classed as a separate type. The frequencies of the various statement types were counted, with assignment statements being by far the most common, and if and goto statements having far lower frequencies but still being far ahead of other types.

- Various special cases of certain statement types were also counted:

    - Assignment statements of the simplest possible kind (e.g. $a = b$ in which b has no arithmetic operators or function calls) were counted, as well as assignments of the form $a = a$ *<op>* α, i.e. those in which the first operand of the source is the same variable as the destination; there exist instructions in many computer instruction sets to make the latter kind of assignment easier to perform than the more general case, if a compiler can take advantage of them. (A large majority of assignments were of the simplest possible kind)!

    - Several special cases of arithmetic operations were counted, as well as occurrences of each different operator: $\alpha + 1$, $\alpha * 2$, $\alpha / 2$, $\alpha ** 2$. These also are easier to perform on many machines than the general cases of addition, multiplication, division, and exponentiation.

- Indexing was examined: the occurrences of variables with no subscripts, one subscript, two subscripts, three subscripts and four subscripts were counted. (About four percent of occurrences of subscripted variables involved more than two dimensions).

- The percentage of do loops using the default increment (one) was measured; and do loops were classified by their length in statements. This last measurement is vaguely useful to designers of hardware instruction-fetch buffers, although the great variability of the number of instructions generated for each Fortran statement by any compiler makes it highly imprecise.

Some dynamic measurements were made, by means of a preprocessor. This program associated a separate counter with every statement in a Fortran program, and added statements to the program to increment each counter when its associated statement was executed, and write the counters out to a file. The breakdown of statements by type, and the breakdown of assignment statements by special cases, were repeated; the dynamic figures showed some significant differences from the static figures. (For instance, the percentage of assignment statements which were simple replacements dropped from 45% to 35%).

The usefulness of these findings to the design of language support systems is clear. Armed with a knowledge of what special cases are likely to occur often, and just how often they occur, the compiler designer (and perhaps also the run-time system designer as well) can make intelligent choices about how code should be generated. We shall see that this kind of knowledge is one of the most useful results of the kind of studies we have examined.

Knuth also makes use of a sampling program to do measurements of actual time spent in portions of the program, rather than frequency counts. This is done by means of a program which, being a supertask of the user program, can interrupt it at regular (or random) intervals and inspect its status. In this case the sampling program looks only at the program counter; with more detailed knowledge of the Fortran run-time system, other interesting data could have been gathered as well.

The sampling program (PROGTIME) does not have the guaranteed accuracy of the frequency count system, of course, *but it does* provide otherwise unavailable information. For instance, Knuth finds that one program spends 70% of its time in two system routines which were involved in input/output editing, although the frequency counts of the source lines which called them were not so high. PROGTIME prints out, normally, a histogram in which each successive interval of 8 words is represented by a bar indicating how often the PC was found there. A much less primitive system would be more useful to users and implementors alike: the addresses should be related to the names of subroutines and functions in the source code, or (even better) to individual line numbers. This, of course, involves some cooperation between the compiler and the sampling program.

2.2. Alexander

Alexander ([17]) studies the XPL system; XPL is a language primarily intended for the implementation of compilers. The raw data obtained are counts of instruction executions, and other information about execution at the instruction level, and thus are useful for evaluation of the System/360 instruction set as well as of the XPL run-time system.

Two methods of gathering data are used; since both of them are very expensive in terms of computer time used, only a limited sample of test programs was studied. The methods are:

- Complete interpretation. Instead of running the compiled program directly on the S/360, it is given as input to a program which interprets the S/360 instructions one at a time. The original interpreter printed a line of data for each instruction executed; due to the great expense of either tabulating or printing the volume of output so produced, Alexander chose to modify the interpreter, to tabulate only the information desired. It took about 200 times as long to run a program on the interpreter as to run it directly.

- Jump tracing. This technique (also see [18]) is a useful compromise, which is not as costly as complete interpretation, but cannot furnish quite as much information, and requires some assistance from the (XPL) compiler. The idea is that straight-line sequences of code are executed at machine speed, but before each branch instruction an instruction is inserted which increments a counter uniquely associated with that branch instruction. Since this instruction must be transparent (i.e. it must not alter the condition codes or registers) and should occupy only a few bytes (because of the lack of addressable core storage), an interrupt instruction was used; the time required by the interrupt handler was fairly high, and a program running with jump tracing took about 44 times as long as the same program running without tracing.

  After the program has finished execution the counters are written out to a file. A separate file has been written by the compiler, containing the information associated with each branch instruction: the instructions in the straight-line code sequence which precedes it and the registers which they use. Subsequent manipulation of the two files can yield much of the same information about register and information usage which was provided by complete interpretation. Clearly much information is missing, as Alexander points out:

  - Information about register contents is lost; thus information about the addresses and lengths of data accesses cannot be gathered. Moreover, information about condition codes is lost as well.

  - The order of the branch instructions is lost; only their counter values are retained. With complete interpretation, Alexander was able to tabulate the frequencies of execution of various pairs and triples of opcodes; but when one of the opcodes is a branch, this information is lost by jump tracing.

  Nevertheless jump tracing is a useful technique for gathering performance information. We shall see a similar technique in the discussion of the CMU Algol monitoring system.

The figures which were computed which are relevant to system performance are:

- Relative frequencies of execution of the various opcodes. Frequencies of pairs and triples of opcodes were also recorded. This information was also gathered statically, i.e. frequencies of occurrence in the object code, rather than of execution, were computed, for comparison.

- Frequencies of use of the various (16) registers. Any instruction may use a register either as an arithmetic accumulator, or as a base or indexing register; these two types of use were counted separately.

Other data were recorded, but those were relevant to the evaluation of the S/360 instruction set, and not to the evaluation of the run-time system. By relating the figures described above to patterns that are known about code generation by the XPL compiler and the coding of the run-time support routines, Alexander was able to draw some conclusions about deficiencies in and potential improvements to the system. For instance:

- 13% of the instructions which were executed were instructions to load a base register, immediately prior to using the base register in a branch instruction. This extremely high percentage reflects badly both on XPL's handling of base registers, and on the architecture of the S/360, which forces the use of registers rather than the program counter for base addresses.

- The N (logical AND) instruction occurs in the string concatenation support routine, and is also generated for condition testing (presumably for the logical AND operator of the language). Its high dynamic frequency of execution, especially in contrast with its low static frequency, indicates that string concatenation is a frequent operation. On the other hand, the low use of register 13, which is used to address all character-string descriptors, at least in comparison with registers 4 to 11 which are used to address the rest of the data area used by a program, leads the author to conclude that "string manipulation is not a major feature of the XPL language". Further study would be needed to reconcile these two observations.

- Registers 1, 2, and 3 are used, in that order, as a "stack" of accumulators to be used for ordinary arithmetic operations. The sharp decrease in dynamic frequency from R1 to R2, and from R2 to R3, confirms Knuth's data indicating that expressions tend to be simple.

- Registers 2 and 3 are used as index registers for array accesses. The dynamic frequencies of instructions which use them as indexes are substantially higher than their static frequencies, leading to the (rather trite) conclusion that array accesses tend to appear more often in loops than outside them.

- Register 15 is reserved as a base register for calls to XPLSM, the "submonitor" which performs I/O for XPL programs. Its extremely low static and dynamic frequencies indicate that in this role it is underutilized, and the extra cost of loading it with the base address for XPLSM before every I/O call would be offset by the benefits of having the register available for other purposes most of the time.

- We have mentioned the high incidence of the L instruction, which is used to load up a base register for a branch instruction. The analysis of instruction pairs indicates that a wide variety of instructions are frequently preceded by L instructions; this suggests that the XPL compiler does not take enough care in code generation to save temporary results in registers, to render subsequent register loading instructions unnecessary.

Probably this is only the tip of an iceberg of useful or at least interesting information that could be deduced from the statistics about opcode and register usage, and other figures that could be tabulated by the interpreter. Here again, however, we are limited by the lack of communication between the data gathering programs and the compiler or run-time system; instead of simply knowing how often the string concatenation routine is called, we must deduce it approximately from the frequency of a rare instruction that it executes. We have no handle at all on some of the other run-time support routines, or other characteristics of the code which are not reflected in opcode or register use frequencies.

A study closely related to Alexander's, but using a different computer, can be found in Wortman ([28]). This involved a computer specifically designed to execute programs in a dialect of PL/I, and the data on frequencies of opcodes were fed back directly to the machine design.

2.3. Batson et al.

Batson et al. ([19],[20]) have studied an aspect of program behavior that has received little systematic observation, namely, the allocation and freeing of storage. This is in the context of the Burroughs B5500 system, in which segments are allocated by requests from the operating system both for program code (one segment per Algol60 block) and for array storage (one segment per row of every array). Actually the second study ignores this structure; "virtual" storage requests are recorded, as if each entire array were allocated one segment, and the group of simple variables declared in each block were one segment.

[19] studies the size distributions of various types of blocks, including free blocks. This study is unique among those in this chapter because there was no selected "sample" group of programs; the entire workload of the Algol 60 system could be studied by suitably instrumenting only the operating system, and indeed since 90% of the workload of the whole system is (Burroughs Extended) Algol 60, the computer system as a whole forms a highly unusual "laboratory" for studying a single language system. The data could be gathered simply by interrupting the system for about one second every so often (usually every two minutes), a performance degradation that was evidently acceptable to, or even not noticed by, users. Since all blocks were linked together in memory, the data gathered and written out during the one-second interrupt is just a list of the links, gotten by scanning linearly through memory. The operating system did overlaying of memory onto secondary storage in units of one segment, and it was suspected at one time that the distribution of sizes of demands for segments might be appreciably different from the distribution of sizes of segments in memory, possibly because segments of particular ranges of size were more frequently overlaid. An altered method of gathering data, that would give a better estimate of the distribution of demands, was devised: the memory would be flushed (all segments

written out to secondary storage); then, some short time later (about 10 seconds), long enough to allow a "reasonable" number of segment requests but before significant overlaying had begun again, the usual 1-second data gathering process was conducted. (It turned out, however, that the equilibrium segment size distributions were not significantly different from the segment demand size distributions).

Distributions were measured for several different kinds of blocks, including free (unallocated) blocks. There are several observations of interest to system designers about these distributions:

- They are peaked very sharply (non-exponentially) at small sizes, with the average segment size ranging from 50 to 150 and the median segment size *always considerably smaller.* The authors point out the unfavorable consequences of this to systems with pages of large fixed size (e.g. 512 words), which are common today.

- The distribution for free blocks was very similar to those for the various types of allocated blocks, indicating a great deal of "checkerboarding" or external fragmentation, presumably a consequence of the design of the dynamic storage allocator used by the operating system.

- The distributions changed in appreciable ways when all allocated blocks generated by "system programs" (i.e. three compilers, and the operating system (Master Control Program) were deleted from the data; the peak for small sizes is much sharper. In this case two thirds of all allocated blocks were less than 30 words in size.

As described earlier, the second study was concerned with a hypothetical scheme of storage demands; in addition, it was desired to gather more data than were available from a simple inspection of segments in memory. Therefore, a number of changes were made to the Algol 60 compiler, the operating system, and even the system hardware to support this experiment, and it was run, not using the daily system workload, but on a set of 34 programs, described as production programs for scientific/engineering applications, covering a wide range of sizes and memory requirements for storage and time.

The compiler was modified to produce code to record the occurrence of events such as block entry, array declarations, and initiation of I/O; the the hardware was modified to include a 1-MHz clock with which the events could be time-stamped; and the operating system was modified to prepare records of the events which could be written onto an external device, and also to record certain events which were outside the ability of the compiled code to instrument. The compiler also generated a file of names connecting the events recorded with various features of the source code.

Static and dynamic distributions are presented, of array segment sizes, contour data (i.e. simple variable) segment sizes, and program segment sizes. In addition, distributions of the lifetimes of the various segments are given, using absolute lifetimes and lifetimes normalized by dividing them by total execution times of the programs.

These distributions have been used to generate stochastic inputs for measuring the behavior of dynamic storage allocation systems by Weinstock ([21]).

2.4. Clark

Clark ([22]) has studied the use and behavior of list structure in (large) Lisp programs. This investigation can only be described as extremely successful, resulting in a wide variety of interesting and useful results, and it is worthwhile to consider what aspects of the methodology or of the system being studied enabled this to happen.

Clark draws a distinction between measurements of snapshots of program execution, called "static", and measurements on traces of execution, called "dynamic" measurements. (Earlier we have used "static" to refer to measurements on source programs, a different distinction). Each of the five large programs in the sample was allowed to run on a task that was long enough and complex enough to cause storage to be garbage-collected and reused several times; at the end of the run, static measurements were made by another program which traversed most (not quite all) of the list structure created by the test program up to that point. Dynamic data were gathered by means of a PDP-10 simulator, which wrote a trace file with an entry for every instruction executed.

The meaningfulness of both the static and dynamic data was enhanced tremendously by knowledge of the data type associated with every pointer in memory. In the Interlisp system studied, this information was particularly easy to obtain, since each page of the address space was devoted to objects of a single data type, and the correspondence between pages and types is available in core during execution.

The problem encountered in the study by Alexander (discussed earlier), that the data gathered by the simulator was hard to relate to the run-time support routines and other primitive actions, was gotten around in this study, by an extraordinarily fortunate circumstance. Each of the important primitive actions studied by Clark (*car*, *cdr*, *rplaca*, *rplacd*, *cons*) is associated with an instruction which is only, and always, executed once by it (respectively, these were hrrz, hlrz, hrrm, hrlm, and pop). This correspondence is equally true whether the Lisp code is compiled or interpreted. It is likely that if, as Clark recommends, the higher-level list-manipulating functions of Lisp are studied in the same detail, data gathering tools more sophisticated than a PDP-10 simulator will be required.

Dynamic measurements were expensive to make: running a program on the PDP-10 simulator took about 60 times as long as running it directly on the PDP-10. Therefore, most dynamic measurements were made on some subset of the five programs, running on relatively small tasks. Some of the dynamic measurements that were made correspond to analogous measurements that were made statically:

- There was a static classification of pointers according to data types of both source and destination. Figures for car pointers and for cdr pointers were kept separate. (As an example, the fact that cdr pointers, in all five programs, point to list structures about three times as often as to nil indicates that the average length of lists is about four cells). The corresponding dynamic measurement was a classification of all references to list structure by the data type referenced. Clark observes that although the static classifications look very similar in all five of the programs used, there is not nearly the same regularity in the dynamic patterns, nor close similarity between static and dynamic patterns for each program.

- "Distances" of pointers, that is, the difference in addresses between the cell containing the pointer and the cell pointed to (if both are list cells), were tabulated. It may not be immediately clear why these figures would be non-random, or why they would be useful. In fact these distances tend to be very small: the distance 1 is by far the most common for both car and cdr pointers, and for both backward (negative difference) and forward pointers, and the number of pointers in any range of distances drops off approximately as the logarithm of distance increases. This can be explained by a combination of two considerations: first, that cells created by successive calls of **cons** are frequently connected by a pointer, and in general many pointers are between cells which were created very close to each other in time; and second, that cells created near to each other in time are likely to be near to each other in space as well. This is especially true at the beginning of execution, or just after a garbage collection, when successive **cons**'s are likely to be adjacent cells; the list of free cells is kept in order of addresses.

  The figures on pointer distances are interesting from the point of view of performance for two reasons:

  - The possibility of a compact encoding of pointers based on distance is tantalizing. Clark discusses several schemes built around the notion that a list pointer could be represented as an offset from the address of the pointing cell, rather than as an absolute address.

  - It is beneficial to the performance of a paging system if lists of cells tend to be cells that are close together, i.e. on the same page. The Interlisp system uses a non-trivial algorithm to find a cell to use for a **cons**, directed at getting the cell created to be on the same page as the cells to which it points. Clark examined the usefulness of this algorithm, by substituting for it a simpler algorithm which simply tried to put each **cons** on the same page as the most recent previous **cons**, and redoing some of the static measurements of pointer distances. The results suggest that the more sophisticated **cons** algorithm makes little difference to the page-locality of pointers.

  Dynamically, the distances of references by **car** and **cdr** were tabulated, giving distributions that were not different in interesting ways from the static distributions.

- The notion of compact encodings can be applied to atom and number pointers as well, and the frequencies of pointers to the various atoms and of occurrences of numbers were tabulated, with the idea of investigating the usefulness of frequency-based encodings of these data types. Among other interesting characteristics of these distributions was that atom pointers approximately followed Zipf's law: the number of pointers to the *n*th most popular atom was proportional to 1 / *n*. The dynamic distributions were markedly different from the static distributions in this case.

Another group of measurements could only be made dynamically:

- A measurement familiar to us from other studies, the simple tabulation of occurrences of the five primitive operations, was done. For all three programs for which this was done, over 80% of all executions of these functions were of car or cdr; slightly more than half the rest were of cons.

- Occurrences of rplaca and rplacd were classified by the types of pointers replaced and the types of the new pointers. (List pointers were sub-classified according to their distances: "adjacent" pointers, "nearby" pointers, and "distant" pointers). This revealed some interesting special cases: for instance, nil was either the replacer or the replaced item in over 80% of the occurrences of rplacd in two of the three programs, and over 60% in the third.

- Another kind of "distance", the distance between two references defined by the number of references that occur between them in time, is of interest because of the widespread use of two-level storage schemes. Clark has used the traces of references to list cells as input to two models of memory management: a cache in which each reference to a list cell causes it to be brought in, and a cache of pages in which each reference to a list cell causes its page (512 words, as defined by the TENEX operating system) to be brought in, both using (for ease of analysis) an LRU replacement algorithm. The figures relevant to hardware and software system designers here are the graphs of "hit ratio" (percentage of references which are to pages already in the cache) against size of cache.

2.5. A general purpose monitoring system

A debugging/analysis facility has been included as a permanent part of the CMU Algol 68 run-time system. This facility will be described here in some detail, to give some understanding of the considerations which affect the usefulness, or lack thereof, of such a facility. We will also describe several experiments which have been conducted using it, and present some of the results from them. The system is described in the terms of the PDP-11 assembly language in which it is written, but the same principles could be applied had it been implemented in any language with suitable conditional compilation features.

We defined by means of macros an "instruction", *mark*, which could be placed anywhere in a code sequence without affecting the execution of the sequence. The effect of a *mark* instruction is that of a subroutine call, except that actions of the subroutine are completely transparent to the rest of the run-time system. Ordinarily the effect of the subroutine is to increment a counter, associated uniquely with the *mark* instruction itself; thus at any point during program execution, it is possible to find out how many times the *mark* instruction has been executed, by interrogating its private counter.

This is the basic feature of the system. What makes it usable is the set of features which keep the system out of the way of non-experimental users, and make the counters easy for experimental users to access and use. Any file containing a *mark* instruction can be assembled so that the instruction assembles to nothing; there is also some run-time control over the counters, so that users can specify that the counter subroutines will do nothing and take the fastest possible return (this is the default). Each *mark* instruction takes an argument which is a five-character name; the system can be told at run-time to print the names of the *mark* instructions as they are executed, and the current value of each counter has been made available, addressed by its name, by using the symbolic debugger available to all HYDRA users. Two variants of the *mark* instruction, called *enter* and *exit*, are used to mark the beginning and end of every significant subroutine in the run-time system; like the printing of the names of the instructions as they go by, this feature is useful primarily for debugging the system rather than for doing experiments.

This is a simple but very general and powerful system for investigating patterns of use of the run-time system. The usual procedure for doing an experiment is as follows:

- Decide what events should be monitored, and what sections of code in the run-time system correspond to each event.

- Put a *mark* instruction with a suitable mnemonic name in each such section of code (if there is not already an *enter* associated with it).

- Assemble the necessary files and link together an experimental run-time system.

- Run the system on selected benchmark programs. At the end of a program run, enter the symbolic debugger interactive system and set down the value of each of the interesting counters. (There is also a feature for writing out all the available counters to a file for later analysis, but we have not made extensive use of this yet).

This system has certain inherent limitations; for instance:

- The information gathered is crude and limited in scope. For instance, we cannot infer the relative patterns of usage of two or more routines, beyond knowing the number of times each was called. If something more complicated than incrementing the associated counter is to be done at each *mark* instruction, a facility exists for having an arbitrary routine executed at all such instructions, but this is sufficiently difficult to use that no such experiments have been done. Thus such information as freqpencies of pairs or triples of mark points, or tabulation of the values of interesting variables at specific mark points, is not available.

- The counters are likely to overflow for long-running programs. This is due to the small word size of the PDP-11. We could have made the counters two words each; but this would have risked the possibility that the counters would not all fit in core. This is due to the small address size of the PDP-11, a closely related problem.

- Since the *mark* instruction is really an interrupt instruction as in the study by Alexander (described earlier), programs run during experiments using this system are noticeably slowed down; we have observed differences of about a factor of 10 in the speeds of "marked" and "unmarked" programs. Therefore this is one of those systems that cannot be let loose on the general user community; experiments must be done on sets of test programs. On the other hand, the factor of 10 is an order of magnitude less than the slow-downs observed in the simulator systems described by Alexander and Clark.

To date, in addition to using this system as a debugging tool, we have used it in three major studies of the behavior of user programs. To further illustrate the strengths and weaknesses of the system, we will describe the studies in detail below.

2.5.1 Effectiveness of various optimizations

The following optimizations have been incorporated in the CMU Algol compiler and run-time system:

- Although in Algol 68 the concepts of "variable" and "pointer" have been united, the compiler distinguishes between them, and generates completely different code for standard operations, such as dereferencing and assignation, on the two types of references. This results in a substantial speed-up in the treatment of variables for some ordinary operations, but there is a tradeoff: it is somewhat more expensive for the user's program to actually use variables as if they were pointers.

- The run-time treatment of multiple values (arrays) is conceptually an extremely elaborate system, designed to avoid the copying of large blocks of storage when multiples or even slices of multiples are dereferenced, assigned, or ascribed during program execution. For instance, a matrix can be passed "by value" as an argument to a procedure call, without causing a copy of it to be made--unless the system detects at some time during the execution of the procedure that a copy must be made to preserve the semantics of the language. Obviously this kind of system involves a tradeoff--a good deal of copying of arrays may be avoided, with a substantial saving in execution time; or a program may require a lot of array copying anyway, so that the overhead required to prevent copying is wasted.

We investigated the usefulness of these optimizations by measuring the frequencies of various types of operations on variables and pointers, and the frequencies of various operations on multiple values which did or did not involve copying. The results are described in some detail in [23], and so will not be further described here.

2.5.2 Behavior of the floating-point arithmetic subroutines

Since hardware to do the basic floating-point arithmetic operations is a non-standard option on PDP-11's, it was necessary to write software to do these. With a view both to making improvements in these routines, and to figuring out which portions of them would be most usefully microcoded, we fully instrumented these routines, inserting enough *mark* instructions to allow the frequency of any possible straight-line sequence of instructions to be measured.

The most interesting of these routines is the one to do floating-point addition and subtraction. Before presenting the results that we have obtained, it is necessary for us to give a description of the action of this routine:

Labels fadd and fsub are separate entries to the same routine; these are the entries used by code produced by the compiler. This routine serves only to put its arguments in suitable format for fxadd and fxsub, described below.

Labels fxadd and fxsub are separate entries to the routine that does the actual floating-point computation; these are the entries used by code in the standard library functions, and by the routines to do complex arithmetic.

Shortcuts to the end of the routine are taken if either argument is zero. The arguments are compared, and the one with the smaller exponent (if the exponents are different) is put in a convenient location (an "accumulator"). The fraction part of the other one is shifted right by from 1 to 24 digits in order to be aligned properly for the addition of the fractions; the shifting is done in the routine labeled shft. (shft is not called if the exponents are equal). (This routine takes a shortcut if the exponents differ by more than 24). The routine shft is complicated by the fact that shifts of 16 digits can be done in one instruction; thus a shift of 15 digits is done by a shift to the left by one digit, followed by a 16 digit right shift. The fraction in the accumulator is negated if the signs of the original arguments were different. The fractions are then added to each other. The result is normalized, either by shifting it to the right by zero or one digit if the fractions were of the same sign, or by shifting it to the left as needed if the fractions were of opposite signs.

The results of the program runs are presented in figure 2. Each column represents a single benchmark program; each row represents a single *mark* instruction, identified by its five-character name at the left. The meanings of the *mark* instructions are as follows:

Name	Circumstances of execution
addrz	call of fadd
subrz	call of fsub

adsbr	call of either fxadd or fxsub
adsb1	2nd operand is nonzero
adsb2	adsb1 executed, but 1st operand is zero
adsb3	1st operand exponent is smaller than or equal to 2nd operand exponent
adsb4	signs of operands are different (Note that subtraction has been turned into addition by negating 2nd operand).
adsb5	result of addition of fractions is negative (Note: see adsb3; exponents must have been equal)
adsb6	high order word of result fraction is zero
adsb7	adsb6 executed, and low order word of result fraction is nonzero
adsb8	result must be normalized by shift to the right (Note: clearly not executed if adsb5, adsb6, or adsb7 are executed)
shftz	call of shft
shft1	shortcut for very small argument taken
shft2	shift of more than 8 digits required
shft3	executed once for each digit of left shift
shft4	executed once for each digit of right shift
shft5	shift of from 9 to 16 digits required
norm1	normalization shift distance is nonzero
norm2	executed once for each digit of normalization

All three benchmark programs are standard matrix manipulation subroutines, taken from [26]. Several interesting conclusions emerge from these figures:

- We might surmise that the normalization loop is the "inner loop" of the addition process, but in fact for the additions in which both arguments are nonzero, the average distance of shift required for normalization (calculated as the ratio of norm2 to (adsb1 - adsb2)) is very low: less than one digit for each of the first two programs, and slightly more than two digits for the third program. A similar conclusion, though not such a strong one, can be drawn about the preliminary shifting loop: for all programs, for additions where the exponents are not equal and thus some shifting is required to align the fractions, the average distance of shift is about two digits.

  This is interesting because it implies that, in fact, the addition routine has no "inner loop". That is, we cannot hope to get a bargain in improving the performance of this process by, say, microcoding, or otherwise streamlining some small portion of it. Moreover, the alignment and normalization loops should be coded, not as "inner loops" are usually coded, to minimize the time per iteration, but rather to minimize the time spent in loop initialization and exit, since the number of iterations is usually so small.

- The alignment loop is conscientiously organized, as explained above, to take advantage of the ease of shifting the fraction by 16 digits at once. Considering the small number of cases in which this advantage was realized (shft2), it is likely that the overhead of detecting these cases is not worth the shifting that is saved. (This hypothesis, of course, must be verified by actually calculating the overhead, from inspection of the code).

- The proportion of additions in which at least one operand is zero ranges from 16% to 32%, an unusually high figure. It is also unusual that the first argument tends to be zero substantially more often than the second argument. Both these observations have consequences for the organization of the beginning of the addition routine, where the two special cases are detected.

A study of floating-point addition closely related to this one can be found in [24].

2.5.3 Behavior of the dynamic storage allocator

The foundation of the CMU Algol run-time system is a dynamic storage allocator (DSA). This largely takes the place of the stack structure used for storage requirements in other implementations of Algol-like languages. For instance, procedure invocation frames, with space for local variables, back links to "outer" invocations, and other environment information, are not sections from a piece of memory organized as a stack, but are gotten by requests to the DSA. It should be clear that the performance of the storage allocator itself is critical to the performance of the system as a whole (see [23]).

Early in the lifetime of the system we realized that the speed of the DSA was not, indeed, all that it should be, and we have sprinkled it with *mark* instructions in an attempt to find out why. As with the last example, we will present some detailed results, for an appreciation of which it is necessary to explain in detail the action of the DSA.

Using the notation of [21] for describing DSA strategies, our system is most closely described by the quintuple: (Q(4,12), Q, R_4, λ, (get,L), λ). Expanding on this notation,

- A (slightly modified) "Quick Fit" scheme is used to organize the class of free storage blocks and to allocate blocks from it. For every size of block from 4 to 12, a separate list is kept, and when a block whose size is in this range is desired, the corresponding "special size list" is examined first (with the exceptions noted below). Only if the special size list is empty, or the block desired is larger than 12 words, does the "general list" get searched (no blocks are smaller than 4 words). The special size lists are in LIFO order; the general list is maintained in LIFO order, and searched for a "first fit".

  In fact this description is not entirely accurate. There are two entries into the system for requesting a block, called **gtbln** and **gtblgen**. The first of these requires (in effect) a block size from 4 to 12, and causes the free lists to be searched as described above. The second of these requires a block size which may be arbitrary, and causes only the general list to be searched. **gtbln** is used when the size required from a block can be deduced from its "type", i.e. the use to which it is to be put. For instance, every multiple value is represented by a block containing pointers to its descriptor and its elements, and all such "multiple value" blocks have the same size. **gtblgen** is used for blocks whose size is not completely determined by their type; examples are invocation frames (since some procedures require more local storage than others), elements blocks of multiples, and descriptor blocks of multiples (since some multiples have more and larger dimensions than others).

Also, there is a separate storage allocation system for real (single precision floating point) number blocks, with a separate special size list and a separate entry, **gtblreal**. The only thing the two allocation systems share is a central common pool of storage, the "free area". When any request for allocation fails because blocks cannot be found on the appropriate free lists, a block of the requested size is chipped from one end or the other of the "free area".

- When a block is found on the general list whose size exceeds that requested by at least 4 words, the excess is split off the original block and returned to the appropriate free list. When the excess is less than 4 words, it is ignored (left with the original block).

  This also is not quite accurate. If gtbln is forced to search the general list, and a block is found with an excess of less than 4 words, the block itself is ignored, i.e. the search continues.

- No "rounding" is done; an attempt is made to get a block of exactly the size requested.

- Adjacent free areas are collapsed only when a request for allocation fails, i.e. a block cannot be found on the usual lists, and the size of the free area has gone to zero. When this happens, a scan through memory finds all free blocks and merges any of them that are adjacent to each other.

- No compaction (relocation of allocated storage) is done.

We can now describe some of the particular inquiries we made into the storage allocation procedure. We inserted *mark* instructions as follows:

Name	Circumstances of execution
coale	call of **coalesce**, the routine to do collapsing of free blocks
gtblg	call of **gtblgen**
gtbln	call of **gtbln**
gtblr	call of **gtblreal**
getfa	call of **getfarea**, the routine to get a block from the free area

The following may be executed during a call to **gtblgen**:

gbglz	executed once for each block examined on the general list
gbgaz	a suitable block is found on the general list (i.e. it is not necessary to call **getfarea**)
gbgbz	gbgaz executed, and the excess size of the block found is too large to ignore (i.e. at least 4 words)
gbgcz	gbgbz or gbnez executed, and the block made from the excess

space is small enough for one of the special size lists

The following may be executed during a call to gtbln:

gbnaz a block is found on the special size list (it is non-empty)
gbnbz executed once for each block examined on the general list (Note: obviously this and the following instructions are not executed when gbnaz has been executed)
gbncz a perfect fit is found on the general list
gbndz a block larger than the request is found on the general list
gbnez gbndz executed, and the excess space is large enough to be separated from the block (i.e. at least 4 words)

The following may be executed during a call to gtblreal:

gbraz a block is found on the real-number list (it is non-empty)
gbrbz a block is gotten from the free area

We also inserted *mark* instructions in the storage deallocation routines, in order to get a breakdown by block types of what blocks were being used. This information will be mentioned below but not given in detail. Figure 3 shows the data gathered while the storage allocator was being monitored. More programs were run for this experiment than for the earlier ones, and there was a wider variety of them:

- hanoi is a program to solve the Towers of Hanoi problem a series of times, with the size of the problem increasing each time (from 4 to 7 disks). Its activity consists mainly of calls to the procedure print, as well as (recursive) calls to the controlling procedure.

- rat2 and rat2* are the same program operating on different input data. The program does matrix decompositions of square matrices, not using real numbers, but instead defining a structure to implement rational numbers and using matrices of rationals. rat2 acts on matrices ranging in size from 1 x 1 to 4 x 4, before aborting due to integer overflow. rat2* goes all the way up to 8 x 8 matrices. As the table shows, its behavior is somewhat different from what simple extrapolation from the behavior of rat2 would suggest.

- ktour finds and prints a knight's tour of an 8 x 8 chessboard.

- det and lsquare are real matrix manipulation programs mentioned in connection with earlier experiments.

These data are extremely interesting from the perspective of trying to estimate the successes and failures of the current DSA system. Here are some of the points of interest:

- Consider the average length to which the general list must be searched before an appropriate block is found by gtblgen. To a first approximation, this is the ratio of gbglz to gtblg. Except for the two runs of rat2, for which this ratio is less than 5, the ratio ranges from 20 to 30. That is, it is in general necessary to search the first 20 to 30 blocks on the general list, before one is found which is large enough to meet the current request.

The behavior that this disastrous figure represents is called "fragmentation". When a request for a medium-size block causes a large block on the general list to be split, the residue block may be in any of several sizes. There is a range of sizes that are too large to be useful on the special size lists, but too small to be useful on the general list; blocks in this range simply hang around on the general list with nothing to do, clogging the front of the list with deadly effect.

- Now consider the spectacular successes of the current DSA system. The percentage of requests to gtbln which are satisfied in just a few instructions, i.e. for which the corresponding special size list is not empty, is the ratio of gbnaz to gtbln. This is never less than 90% and seems to average about 95%. The corresponding ratio for the real-number list, for the two programs which use it, is even higher (it is the ratio of gbraz to gtblr). Moreover, when the special size lists do fail, the resulting search of the general list stops after about 1 or 2 blocks. (Blocks small enough to be on the special size lists do occasionally find their way to the general list, e.g. elements blocks for small arrays, or blocks for short strings). (The average search length is the ratio of gbnbz to (gbnaz - gtbln). Thus the method of keeping special size lists, when it can be applied, is extremely useful -- it can reduce storage allocation overheads to near their minimum values.

- The breakdown of block types, not presented above, yielded other useful information. One type of block, the invocation frame, seemed to dominate the others in frequency of use -- only for det and lsquare was the number of deallocations of invocation frames less than 40% of the total number of deallocations (for these two programs the proportion was closer to 20%). For those programs that used them at all, the other varying-size blocks (elements blocks, descriptor blocks, structured value blocks, strings) comprised large proportions of the total allocation-deallocation traffic. For instance, about 17% of the deallocations done during rat2* are of structured value blocks.

A number of ideas for improving the DSA system have suggested themselves to us, and we can use these data to make preliminary appraisals of them. The most promising one appears to be the extension, to the maximum degree possible, of the idea of "special size lists". In the improved system these would not be limited to a known set of sizes, fixed for all programs, as in the present system, but would include lists tailored to the needs of each particular program. For instance,

- All invocation frames for a particular procedure are equal in size (since the number of arguments and the maximum potential requirement for local storage do not vary from one invocation to the next); so each procedure has its own special free list, from which frames for invocations of it are allocated.

- All instances of a particular structured mode have the same size; so each such mode has its own free list.

- All descriptors for arrays of one dimension have the same size; similarly for arrays of two dimensions, three dimensions, and so on. There are separate free lists for descriptors for each of the first few (commonly used) dimensions; descriptors for really high-dimensional multiples revert to the general storage list.

In future documents we will report on the results of experimentation with this and other improvements to the DSA system.

2.6. Summary

What can we learn, from these experiences, about methodology? What should a system implementor do, and know, in order to have adequate tools for understanding the behavior of a language system?

We have seen that machine simulators, such as the S/360 simulator used by Alexander and the PDP-10 simulator used by Clark, are far from being adequate to the purpose. They are prohibitively expensive to run on test programs of interesting size; at the same time they cannot ordinarily gather all the information that is desired. More sophisticated measurement tools will have to be devised, and they will have to be designed along with, and into, the language systems themselves. A measurement tool must "know" about the structure of the thing being measured. Events which are significant at the source level (such as transitions from one statement to the next or from one line to the next, procedure entries and exits, decision points in loops and conditionals), and events which are significant at the implementation level (such as entry and exit of run-time support routines, or even execution of arbitrary pieces of code in the run-time support system), must be reflected to the measurement program as events that it can record; the use of indirect event records, such as executions of unusual instructions, can only be a stopgap measure.

We should also consider another problem which has not been solved by any of these studies in an entirely satisfactory manner, namely, the selection of test programs. As we have seen from several of the studies, different application areas and even different programs within one area can show remarkably different characteristics at execution time; and yet the selection of programs for the samples in these studies has been in all cases almost haphazard, and was not even discussed by some of the authors. This is a subject that badly needs attention more serious than that which it has so far been given.

References

[1] Svobodova, L., "Computer Performance Measurement and Evaluation Methods: Analysis and Applications", American Elsevier, 1976.

[2] Wichmann, B.A., "A comparison of Algol 60 execution speeds", National Physical Laboratory Report CCU3, Jan. 1969.

[3] Wichmann, B.A., "Algol 60 Compilation and Assessment", APIC Studies in Data Processing, 10, Academic Press 1970.

[4] Wichmann, B.A., "Basic Statement Times for Algol 60", National Physical Laboratory Report NAC 42, 1973.

[5] Hibbard, P.G., "The Run-time Organisation of an ALGOL 68 Sublanguage Compiler", in Proceedings of an International Conference on Algol 68 Implementation, P. King (ed.), Utilitas Mathematica, Winnipeg 1974, 19-25.

[6] Wichmann, B.A., "Some Statistics from ALGOL Programs", National Physical Laboratory Report CCU11, Aug. 1970.

[7] Wichmann, B.A., "Ackermann's Function: A Study in the Efficiency of Calling Procedures", BIT 16, 1 (1976), 103-110.

[8] Wichmann, B.A., "How to Call Procedures, or Second Thoughts on Ackermann's Function", Software--Practice and Experience 7 (June-July 1977), 317-329.

[9] Sundblad, Y., "The Ackermann Function. A Theoretical, Computational, and Formula Manipulative Study", BIT 11 (1971), 107-119.

[10] Boom, H.J., and E. deJong, "A Critical Comparison of Several Implementations of Programming Languages", Report IW 58/76, Mathematisch Centrum, Amsterdam, 1976.

[11] Curnow, H.J., and B.A. Wichmann, "A Synthetic Benchmark", The Computer Journal 19, 1 (1976), 43-49.

[12] Morgan, D.E., and J.A. Campbell, "An Answer to a User's Plea?", Proc. First Annual SIGME Symposium on Measurement and Evaluation, Feb. 1973, 112-120.

[13] Randell, B., and L.J. Russell, "Algol 60 Implementation", Academic Press, London, 1964.

[14] Gibson, J.C., "The Gibson Mix", IBM TR 00.2043, June 1970.

[15] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software--Practice and Experience 1, 2 (1971), 105-134.

[16] Elshoff, J.L., "A Numerical Profile of Commercial PL/I Programs", Software--Practice and Experience 6 (1976), 505-525.

[17] Alexander, W.G., "How a Programming Language is Used", University of Toronto Technical Report CSRG-10, Feb. 1972.

[18] Gaines, R.S., "The Debugging of Computer Programs", Ph.D. Thesis, Princeton University, 1969 (Ch. 3).

[19] Batson, A.P., S.-M. Ju, and D.C. Wood, "Measurements of Segment Size", CACM 13, 3 (1970), 155-159.

[20] Batson, A.P., and R.E. Brundage, "Segment Sizes and Lifetimes in Algol 60 Programs", CACM 20, 1 (1977), 36-44.

[21] Weinstock, C.B., "Dynamic Storage Allocation Techniques", Ph.D. Thesis, CMU, 1976.

[22] Clark, D.W., "List Structure: Measurements, Algorithms, and Encodings", Ph.D. Thesis, CMU, 1976.

[23] Hibbard, P.G., P. Knueven, and B.W. Leverett, "A Stackless Run-time Implementation Scheme", in Proceedings of the Fourth International Conference on the Implementation and Design of Algorithmic Languages, R.B.K. Dewar (ed.), Courant Institute, 1976, 176-192.

[24] Sweeney, D.W., "An Analysis of Floating-point Addition", IBM Systems Journal 4, 1 (1965), 31-42.

[25] Bloom, B.H., C.G. Feldman, M.H. Clark, and R.K. Coe, "Criteria for Evaluating the Performance of Compilers", Rome Air Development Center report RADC-TR-74-259, Oct. 1974.

[26] Grune, D., "The MC Algol 68 Test Set", Report IW 53/75, Mathematisch Centrum, Amsterdam, 1975.

[27] van Wijngaarden, A. (ed.), "Revised Report on the Algorithmic Language Algol 68", Acta Informatica 5, 1-3 (1976).

[28] Wortman, D.B., "A Study of Language Directed Computer Design", Ph.D. Thesis, Stanford University, 1973.

Appendix A: Some basic statements benchmark results

Figure 1 presents the original list of basic statements from [4], with the Algol 68 equivalents for them that were used when the program was run on the CMU Algol 68 system, and the average time for each statement in microseconds. Of the identifiers used in the various statements, *x*, *y*, and *z* are real variables initialized to 1.0, *k*, *l*, and *m* are integer variables initialized to 1, *e*1, *e*2, and *e*3 are integer arrays, and *p*0, *p*1, *p*2, and *p*3 are procedures which do nothing and return no result.

The nature of this table is such that it requires substantial commentary!

- The compiled code includes calis to a "line number change" routine. These are placed at various landmark points, e.g. between statements and at the beginning of loop bodies, whenever the compiler detects that the source line number has changed since the last previous landmark point; the routine updates the record kept at execution time of the source line number, as well as performing a few other functions, such as checking for pending software interrupts from other processes in multiprocess systems.

  It was desired to make these timings without the calls to the line number change routine. However, at the time the benchmark program was run, the compiler could not be prevented from generating the calls, so the routine itself was patched to return without doing anything.

- The times given certainly cannot be presumed to be as accurate as they are given to be. Even the times for the simplest statements show some glaring indications of error. For instance, the times for $x := 1$ and $x := l$ differ by about 10 microseconds, although examination of the code for these two indicates that they differ only in one addressing mode in one instruction, i.e. the times should differ by not more than 2 microseconds. Earlier versions of these figures even contained internal inconsistencies, e.g. the time for $x := y \uparrow 2$ was greater than that for $x := y \uparrow 3$ although less code was executed for the former statement. Internal inconsistencies are not an uncommon occurrence in the many sets of figures obtained for this benchmark program on various systems.

- For some of the Algol 60 statements it was difficult or impossible to find Algol 68 equivalents:

  - The semantics of exponentiation in Algol 60 require that the for the statement $x := y \uparrow z$, the logarithm of y should be computed, and multiplied by z; e should be raised to the power of the product to get the result. Thus the logical Algol 68 equivalent of this statement, and the one used in [10], is $x := exp\ (y * ln\ (z))$. The equivalent used here, on the other hand, is probably not a very good choice, since exponentiation to an integer power is a completely different operation.

  - The switch construct has no obvious equivalent in Algol 68, and we have used the suggestion from the Revised Report on Algol 68 ([27]) of using a procedure whose body is a **case** clause, to perform the equivalent function.

Naturally this contraption is seldom seen in Algol 68 programs, making it a rather poor choice for a benchmark program.

  - The Algol 68 operators **sign** and **entier** yield integer values, and thus do not correspond very well to their Algol 60 counterparts. For instance, the time for the statement x := **sign** y is dominated by the time required for the conversion from integer to real.

- As Wichmann points out, some of the statements are likely to take less time than they would, if their inputs were more typical. For this system, the only unusually low figure is for x := *ln* (y); the natural logarithm function takes a shortcut for arguments sufficiently close to 1.0.

- The following characteristics of the system make the raw numbers somewhat more intelligible:

  - All arrays are initialized when declared (all elements are set to an "undefined" value that is recognized by the system and can neither be mistaken for a legal value nor arrived at as a result of normal arithmetic computation).

  - Array address computations, and conversions from integer to real, and other operations which, when their arguments are known at compile time, might be done at compile time in some systems, are all done at execution time in this system.

  - Almost all operations of any complexity, including most assignations in the benchmark program, are done by calls of out-of-line subroutines. The only noteworthy exception is that assignations involving integers (such as k := l) are done in line.

  - The PDP-11 computers on which the test was conducted did not have hardware to do floating point arithmetic, nor to do integer multiplication and division.

- The times for the statements involving procedure calls are unusually high. Actual instruction counts had led us to expect much smaller times. This was one of the observations that led us to suspect odd behavior of the dynamic storage allocator, as described in section 2.5.3.

Figure 1

Basic Statement	Algol 68 version	time in microseconds
x := 1.0	x := 1.0	63.0
x := 1	x := 1	288.2
x := y	x := y	64.5
x := y + z	x := y + z	448.3
x := y * z	x := y * z	539.4
x := y / z	x := y / z	469.2
k := 1	k := 1	23.3
k := 1.0	k := ROUND 1.0	148.7
k := l + m	k := l + m	72.8
k := l * m	k := l * m	207.7
k := l DIV m	k := l % m	198.3
k := l	k := l	23.5
x := l	x := l	278.4
l := y	l := ROUND y	145.0
x := y ↑ 2	x := y ↑ 2	909.6
x := y ↑ 3	x := y ↑ 3	926.6
x := y ↑ z	x := y ↑ m	245.4
e1 [1] := 1	e1 [1] := 1	465.2
e2 [1, 1] := 1	e2 [1, 1] := 1	655.9
e3 [1, 1, 1] := 1	e3 [1, 1, 1] := 1	898.7
l := e1 [1]	l := e1 [1]	379.3

BEGIN
REAL a;
END
	BEGIN
	REAL a;
	SKIP
	END
		350.6

BEGIN
ARRAY a[1:1];
END
	[1:1] REAL a;
	SKIP
		1958.8

BEGIN
ARRAY a[1:500];
END
	[1:500] REAL a;
	SKIP
		13369.8

BEGIN
ARRAY a[1:1,1:1];
END
	[1:1,1:1] REAL a;
	SKIP
		2428.5

BEGIN
ARRAY a[1:1,1:1,1:1];
END
	[1:1,1:1,1:1] REAL a;
	SKIP
		5417.8

GOTO x;
x:;
	GOTO x;
	x:SKIP;
		1272.9

BEGIN
SWITCH ss := pq;
GOTO ss [1];
pq: ;
	PROC ss =
	(INT i) VOID:
	CASE i IN GOTO pq, SKIP OUT;
	ss (1);
	pq: SKIP
		2890.75

END

x := sin (y)	x := sin (y)	7280.1
x := cos (y)	x := cos (y)	7583.3
x := abs (y)	x := ABS y	103.5
x := exp (y)	x := exp (y)	6921.7
x := ln (y)	x := ln (y)	2184.6
x := sqrt (y)	x := sqrt (y)	2531.5
x := arc tan (y)	x := arc tan (y)	8456.0
x := sign (y)	x := SIGN y	353.3
x := entier (y)	x := ENTIER y	401.5
p0	p0	880.4
p1 (x)	p1 (x)	1007.8
p2 (x, y)	p2 (x, y)	1060.8
p3 (x, y, z)	p3 (x, y, z)	1111.2

Figure 2

	lsquare	det	choleski
addrz	720	540	582
subrz	120	204	312
adsbr	840	744	894
adsbl	828	680	842
adsb2	119	176	176
adsb3	431	58	436
adsb4	296	140	481
adsb5	46	40	113
adsb6	3	0	24
adsb7	3	0	24
adsb8	157	120	55
shftz	554	446	413
shftl	0	0	0
shft2	10	0	45
shft3	36	0	196
shft4	1097	933	818
shft5	7	0	40
norml	221	140	405
norm2	494	214	1425

Figure 3

	hanoi	rat2	det	ktour	rat2*	lsquare
coale	1	1	1	1	20	1
gtblg	2853	1896	1349	1141	13554	1419
gtbln	3554	597	2061	353	2997	1195
gtblr	0	0	3529	0	0	4294
gtblp	2	2	2	2	2	2
getfa	223	219	194	170	255	198
gbglz	59002	11777	42221	40002	83834	43148
gbgaz	2632	1703	1180	985	13141	1245
gbgbz	633	999	754	622	9057	706
gbgcz	243	263	245	106	1798	227
gbnaz	3547	539	2025	332	2857	1158
gbnbz	7	82	54	28	203	50
gbncz	0	6	0	0	26	0
gbndz	6	34	26	15	100	22
gbnez	6	27	12	9	88	14
gbraz	0	0	3520	0	0	4276
gbrbz	0	0	9	0	0	18